Visible Nearest Neighbor Queries

Sarana Nutanong[1], Egemen Tanin[1,2], and Rui Zhang[1]

[1] Department of Computer Science and Software Engineering,
University of Melbourne, Victoria, Australia
[2] NICTA Victoria Laboratory, Australia

Abstract. We introduce the visible k nearest neighbor (VkNN) query, which finds the k nearest objects that are visible to a query point. We also propose an algorithm to efficiently process the VkNN query. We compute the visible neighbors incrementally as we enlarge the search space. Our algorithm dramatically reduces the search cost compared to existing methods that require the computation of the visibility of all objects in advance. With extensive experiments, we show that our algorithm to process the VkNN query outperform the existing algorithms significantly.

Keywords

Nearest Neighbor, Spatial Algorithms, Spatial Data Structures

1 Introduction

In many interactive spatial applications, users are only interested in objects that are visible to them. For example, tourists are interested in locations where a view of a scene, e.g., sea or mountains, is available; in an interactive online game, a player is commonly interested in having an overview map of the enemy locations that can be seen from his/her position. A simple visible nearest neighbor (VNN) query finds the nearest object that is visible to a query point. Figure 1 shows an example of the VNN query. The dataset consists of data objects (black dots and circles) and obstacles (lines). Q is the query point. From Q, some objects are visible (b, d, f, h, represented by black dots) while the views of some objects are blocked by obstacles, namely, invisible (a, c, e, g, represented by circles). a has the smallest distance to Q among all objects, therefore a is the NN in the traditional sense. However, a is not the VNN of Q since a is invisible to Q. Among all the visible objects, b is nearest to Q, therefore b is the VNN of Q. In analogy to the k nearest neighbor (kNN) query, we can have the visible k nearest neighbor query. In the example, the V3NN of Q is $\{b, d, f\}$.

Formally, the VNN query is defined as follows:

Definition 1 (Visible nearest neighbor (VNN) Query) *Given a data set S and a query point Q, find an object $O \in S$, so that: (1) O is visible to Q; and (2) $\forall O' \in S$, if O' is visible to Q, then $dist(O, Q) \leq dist(O', Q)$, where $dist()$ is a function to return the distance*[3] *between the query point and an object. O is called the visible nearest neighbor (VNN) of Q.*

Further, the definition of the VkNN query is given as follows:

Definition 2 (Visible k nearest neighbor (VkNN) Query) *Given a data set S and a query point Q, find a set of objects A, so that: (1) A contains k objects from S; and (2) $\forall O \in A$, O is visible to Q; and (3) $\forall O' \in S - A$, if O' is visible to Q, then $dist(O, Q) \leq dist(O', Q)$, where $dist()$ is a function to return the distance between the query point and an object. A is called the visible k nearest neighbor set (VkNN) of Q.*

The notion of visibility has been extensively studied in the area of computer graphics and computational geometry [1]. Specifically, given a set of spatial objects and obstacles, there are efficient algorithms to determine the visible regions, i.e., the regions where objects are visible. Therefore, a naive method to process the VkNN query is to use a NN search algorithm to find objects progressively according to their distances

[3] In this paper, we focus on the Euclidian distance, although any distance function can be used in general.

to the query point and use the visibility knowledge as the post-condition to discard the invisible objects. The process stops when k visible objects are retrieved. But it requires the visible regions to be determined in advance, which accesses all the obstacles.

In this paper, we propose an algorithm for VkNN search without pre-computing the visible regions. Our algorithm is based on the observation that a farther object cannot effect the visibility of a nearer object. Therefore, we can start from retrieving the nearest object and incrementally obtain the knowledge of visibility while finding visible neighbors. By this means, the cost for determining the visibility of data objects is minimized.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 describes our proposed VkNN algorithm. Section 4 presents the results of our experimental study and Section 5 concludes the paper.

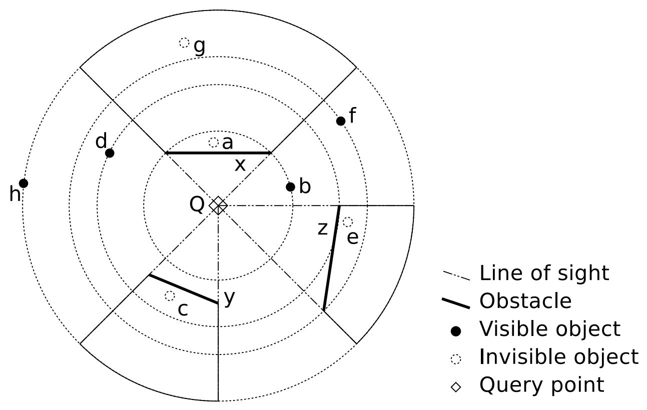


Fig. 1. The VNN query

2 Related Work

An incremental nearest neighbor algorithm was introduced in [6] based on a spatial index such as the R-tree [5]. Using this incremental approach, the number of nearest neighbors required, k, need not to be specified in advance and can be retrieved in order incrementally. The major benefit of this algorithm compared to the kNN approach in [8] is that obtaining the next nearest neighbor after having obtained the k nearest neighbors incurs minimal additional effort. The algorithm uses a priority queue to maintain the candidates (index nodes or data entries) ranked by an optimistic distance estimator. Every time an object is retrieved from the priority queue, the property of the optimistic distance estimator ensures that nothing left in the priority queue can be nearer to the query point than the object. Therefore, the objects are retrieved incrementally in increasing order of their distances to the query point. Our VkNN uses the same incremental search strategy to find the results. More importantly, we incrementally compute the visibility knowledge, therefore the visibility determination cost is greatly reduced.

The notion of the constrained nearest neighbor query which combines the NN problem with the linear-constrained-search problem is introduced in [4]. A constrained nearest neighbor query finds the nearest neighbor inside a convex-polygonal spatial constraint. Only nearest neighbors satisfying the constraint set are returned and only regions satisfying the constraints are explored. Although the visibility knowledge can be modeled as a set of constraints, it is inefficient to use the constraint nearest neighbor algorithm to solve the VkNN problem. This is because the constraint nearest neighbor query requires the constraints to be given in advance causing preprocessing of the visibility constraints.

Recently, a new type of spatial query, nearest surrounder (NS), is presented in [7]. A nearest surrounder query finds the nearest object for each distinct range of angles around the query point. Each returned object is associated with its orientation, which is the range of angles in which the object is the nearest to the query point. By this means, only objects that are not entirely eclipsed by nearer objects can be returned as results. This is similar to VkNN which finds k visible objects around the query points. The main difference of these two spatial queries is that NS finds all “visible” objects around the query point whereas the number of objects in VkNN is user-determined. Using our *pre-pruning* mechanism, we can modify VkNN to efficiently find the complete set of nearest surrounders by running VkNN to completion and associating each returned object with its visible range of angles.

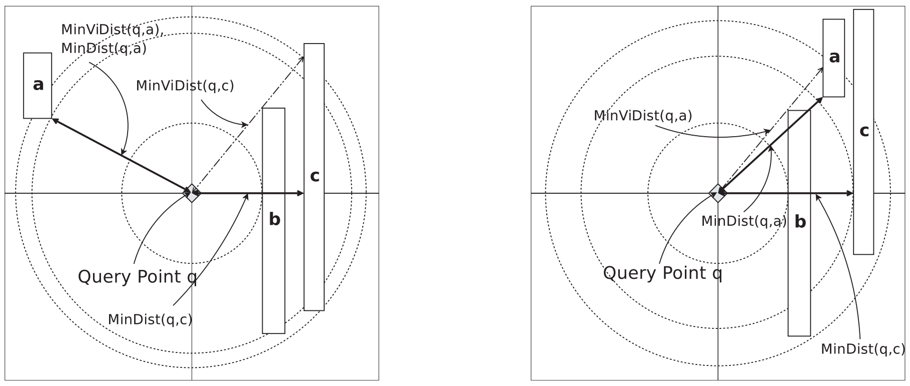
3 Visible Nearest Neighbor Algorithms

Our algorithm is based on the observation that a farther object cannot effect the visibility of a nearer object. Therefore, both results and visibility knowledge can be incrementally obtained. By doing this, the cost for determining the visibility of data objects is minimized.

Without loss of generality, to simplify our discussion, we make no distinction between the point objects which can be returned as results and the obstacles that create invisible regions of space; we refer to both of them as *objects*. In general, objects are represented by polygons (points are special cases of polygons). Objects represented as polygons (i.e., with extents) can be partially visible. Therefore, we introduce a new distance function, called *MinViDist*, which returns the distance between a query point and the nearest visible point of an object with regard to a given visibility setting. This distance function is different from the commonly used MinDist function. The MinDist between a polygon and a point is the distance between the point and the nearest point in the polygon.

Formally, the MinViDist function is defined as follows:

Definition 3 (MinViDist) *Given a polygon P and a query point Q, MinViDist is the distance between Q and a point $T \in P$, so that (1) T is visible to Q; and (2) $\forall T' \in P$, if T' is visible to Q, then $dist(T, Q) \leq dist(T', Q)$, where $dist()$ is a function to return the distance between two points.*



(a) (b)

Fig. 2. Comparison of MinDist and MinViDist

As shown in Fig. 2(a), the MinViDist between the query point q and the object c is the length of the dashed line pointed to the surface of c, whereas the MinDist between q and c is the length of the solid line pointed to the surface of c, passing through the obstacle b. The MinDist and MinViDist for object a are the same since no obstacle is in between. If we order the objects according to MinDist, we get $\{b, c, a\}$; if we order the objects according to MinViDist, we get $\{b, a, c\}$. Therefore, if we rank the objects according to MinDist, we may not get the answers for a VkNN query in the correct order.

Figure 2(b) shows another example of the difference between MinDist and MinViDist. The MinViDist between the query point q and the object a is the length of the dashed line pointed to the surface of a, whereas the MinDist between q and a is the length of the solid line pointed to the surface of a. They are different because of the obstacle b. The MinDist between q and c is the length of the solid line pointed to the surface of c, passing through the obstacle b; whereas the MinViDist between q and c is infinity, that is, c is invisible to q. If we issue a 3NN query (which actually uses MinDist to rank the objects), we get $\{b, c, a\}$; if we issue a V3NN query (which actually uses MinViDist to rank the objects), we get $\{b, a\}$. c is not returned for V3NN since it is invisible. Therefore, using MinDist, we may not even get the same set of answers as those retrieved according to MinViDist.

Given the definition of MinViDist, we can now describe our algorithm to process the VkNN query. In our presentation, we assume that all the objects are indexed in an R-tree [5], although our algorithms are applicable to any hierarchical spatial index structure such as the k-d-tree [3] or the quadtree [9]. We propose three variations of the algorithm which differ in whether pruning objects by visibility is done before or after retrieving a node, and differ in what distance estimator we use to order objects in the priority queue that maintains the candidates. The results are ranked according to the MinViDist function for all of three variations.

```
ALGORITHM PrePruning-MinDist (IsNewQuery, QueryPoint)
if IsNewQuery then
    PriorityQueue ← PriorityQueueCreate()
    VisibilityConstraintSet ← VisibilityConstraintSetCreate(QueryPoint)
    NewPQueueNode ← PQueueNodeCreate(RootNode)
    PriorityQueue.Insert(NewPQueueNode)
endif
while PriorityQueue.IsEmpty() = false do
    PQueueNode ← PriorityQueue.Front()
    SpatialEntity ← PQueueNode.SpatialEntity
    PriorityQueue.PopFront()
    if IsInvisible(VisibilityConstraintSet, SpatialEntity) then
        continue
    else if IsObject(SpatialEntity) then
        PQueueNode.Distance ← MinViDist(VisibilityConstraintSet, QueryPoint, SpatialEntity)
        if PQueueNode.Distance < PriorityQueue.Front().Distance then
            Update(VisibilityConstraintSet, SpatialEntity)
            return (SpatialEntity, PQueueNode.Distance)
        else
            PriorityQueue.Insert(PQueueNode)
        endif
    else if IsBlock(SpatialEntity) then
        for SubEntity in SpatialEntity.SubEntities do
            if IsVisible(VisibilityConstraintSet, SpatialEntity) then
                NewPQueueNode ← NewPQueueNodeCreate()
                NewPQueueNode.Distance ← MinDist(QueryPoint, SpatialEntity)
                PriorityQueue.Insert(NewPQueueNode)
            endif
        endfor
    endif
endwhile
return nil
```

Fig. 3. Algorithm **PrePruning-MinDist**

PostPruning: We use a similar kNN search algorithm as in [6] (MinDist is used to sort the candidates in the priority queue). The only differences are the post-pruning process discarding invisible objects and result-ranking use MinViDist. This variation of the algorithm is an adaptation of the existing algorithm [6].

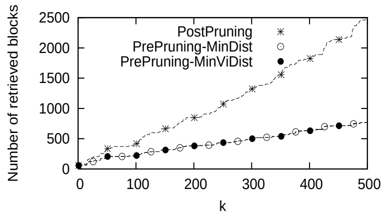
PrePruning-MinDist: Similar to PostPruning, MinDist is used as the estimator but the index nodes and objects are checked for visibility before they are retrieved. By this means, we avoid needlessly searching invisible regions which will not produce useful results. Figure 3 shows the detailed steps of the algorithm. The algorithm is also based on the incremental NN algorithm [6]. The differences between the two algorithms are that: (1) index nodes and objects are checked for visibility before they are retrieved from the R-tree (Line 23) and after they are dequeued from the head of the priority queue (Line 11); and (2) results are ranked according to MinViDist (Line 14).

PrePruning-MinViDist: This variation differs from PrePruning-MinDist only in that we use a MinViDist (which is more accurate and more expensive to calculate) as the metric to order the candidates in the candidate priority queue. This is done by replacing MinDist in Fig. 3 Line 25 with MinViDist.

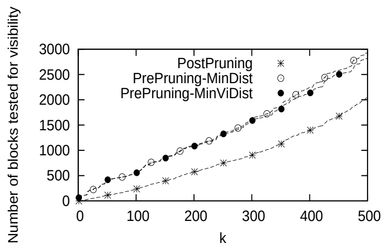
4 Performance Evaluation

This section compares the performance of the three variations of the VkNN search algorithm described in Section 3. In our implementation, a disked-based R*-tree [2] is used. The data set contains 10,000 objects synthetically generated and uniformly distributed in a unit 2-dimensional space. These 10,000 objects also serve as obstacles. The width and height of each object are randomly generated in a uniform manner ranging between 0.0001 and 0.001 units. The fanout of the R*-tree nodes is 24. The performance evaluation is conducted on a Intel Pentium 4 machine with the main memory of 2 GB.

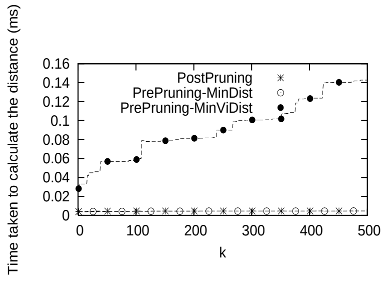
The cost for VkNN calculation can be broken down into five components: data retrieval, visibility determination, distance calculation, priority queue access, and reevaluation (the cost for reinserting objects back into the priority queue for reevaluation). This breakdown can be used to determine which of the three approaches presented in Section 3 is more suitable in different settings.



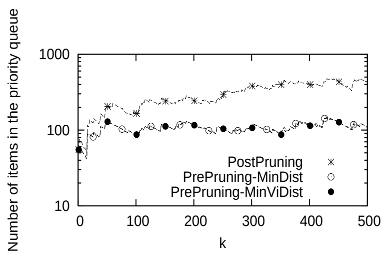
(a) Data retrieval



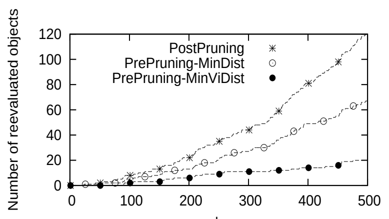
(b) Visibility determination



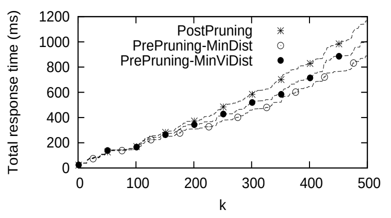
(c) Distance calculation



(d) Queue Size



(e) Reevaluation



(f) Total response time

Fig. 4. Experimental Results

The experimental results are presented in six charts. Each chart plots the performance of the three approaches as a function of k. Figure 4(a) compares the data-retrieval costs of the three variations, which is measured by the number of blocks accessed. The result for visibility-determination costs is given in Fig. 4(b). Figure 4(c) shows the time to calculate the distance of a node in microseconds. Fig. 4(d) presents the maintenance cost of the priority queue in terms of the queue size. It is plotted on the logarithmic scale because the priority-queue access cost is logarithmic with respect to the queue size (when implemented using heaps). Figure 4(e) displays the reevaluation cost measured by the number of objects reinserted into the priority queue. This incurs a different type of cost from the data retrieval cost because it does not involve storage access. The total response time is displayed in Fig. 4(f).

These experimental results suggest that PrePruning-MinDist have the same data-retrieval and priority-queue maintenance costs as PrePruning-MinViDist but the time taken to calculate the distance of each block for PostPruning and PrePruning-MinDist is much smaller (the reevaluated object counts are small in general for our settings and can be ignored). Due to the fact that PostPruning has a larger search space, it has higher reevaluation and queue processing costs. According to the results, the improvement in terms of search-space reduction made by using MinViDist instead of MinDist as the distance estimator is considered to be insignificant while the cost for MinViDist calculation is much higher than MinDist calculation. We suspect the reason that PrePruning-MinDist and PrePruning-MinViDist incur the same amount of search space is that the R*-tree margin-minimization criterion [2] prefers quadratic-shaped index nodes. This means that most nodes are either invisible or have MinDist about the same as MinViDist; instead of having a large difference in MinDist and MinViDist. We can therefore conclude that PrePruning-MinDist which uses the MinDist metric as the distance estimator and visibility pruning is a better approach for VNN. The MinViDist metric should be primarily considered for ranking purposes than as a search-estimator. In disk-based R*-trees, PrePruning-MinDist is clearly a better option than PostPruning. Since in such settings, the data-retrieval costs dominates the total cost of query processing [6]. A setting that would make the first approach comparable to PrePruning-MinDist is when the R*-tree is memory-based and the CPU speeds are very slow, because this makes it cheaper to retrieve a block from the R*-tree but more expensive to determine the visibility of a block.

5 Conclusion

In this paper, we introduced a new type query, the *visible k nearest neighbor* (VkNN) query. We also introduced a new metric called *minimum visible distance* (MinViDist) for result ranking as well as search ordering. Furthermore, we propose an algorithm (particularly, three variations of the algorithm) to process the VkNN query. All the three variations build up the visibility knowledge incrementally as the visible nearest objects are retrieved and therefore the computation cost for visibility determination is minimized. It is shown in the experimental results that the latter two variations, PrePruning-MinDist and PrePruning-MinViDist, put more effort on the visibility pruning in order to reduce the data-retrieval cost. This could be beneficial in settings with disk-based or network-based storage where the data-retrieval costs are more critical than the CPU costs. These two variations differ in the computation cost of calculating the distance estimator and the number of disk accesses for visible objects, which is a tradeoff depending on computing power and object retrieval cost. Both of them are more efficient than the first variation, PostPruning, which is an adaptation of the existing algorithm [6]. In our experiments, the improvement in terms of response time of the query processing is up to 35%.

References

1. T. Asano, S. K. Ghosh, and T. C. Shermer. *Visibility in the plane*, pages 829–876. Handbook of Computation Geometry. Elsevier Science Publishers, Amsterdam, The Netherlands, 2000.
2. N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *Proceedings of the ACM SIGMOD Conf.*, pages 322–331, Atlantic City, NJ, USA, 1990. ACM Press.
3. J. L. Bentley. Multidimensional binary search trees used for associative searching. *CACM*, 18(9):509–517, 1975.
4. H. Ferhatosmanoglu, I. Stanoi, D. Agrawal, and A. El Abbadi. Constrained nearest neighbor queries. In *Proceedings of the SSTD Conf.*, pages 257–278, London, UK, 2001. Springer-Verlag.
5. A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD Conf.*, pages 47–57, Boston, MA, USA, 1984. ACM Press.
6. G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM Trans. Database Syst.*, 24(2):265–318, 1999.
7. K. C. K. Lee, W. C. Lee, and H. V. Leong. Nearest surrounder queries. In *Proceedings of the ICDE Conf.*, pages 85–94, Atlanta, GA, USA, 2006. IEEE Computer Society.
8. N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *Proceedings of the ACM SIGMOD Conf.*, pages 71–79, San Jose, CA, USA, 1995. ACM Press.
9. H. Samet. The quadtree and related hierarchical data structures. *ACM Comput. Surv.*, 16(2):187–260, 1984.